The Virtual Processor Interface: Linux Kernel Support for User-level Thread Systems

Gregory D. Benson, Matthew Butner, Shaun Padden, and Alex Fedosov
Keck Cluster Research Group
Department of Computer Science
University of San Francisco
{benson,mbutner,spadden,afedosov}@cs.usfca.edu

Abstract

Despite an increasing need for thread support in language run-time systems and parallel libraries such as in Java and OpenMP, there is limited support for custom, multiprocessor capable, user-level thread systems in the Linux kernel. To address this lack of support we have developed the virtual processor interface (VPI) for Linux. Our VPI implementation consists of a small set of kernel modifications and new system calls combined with a small user-level library that provide an interface that can be used to build thread systems. VPI uses a form of scheduler activations so that user-level thread systems can have complete control over the amount of parallelism for an application and the scheduling of threads onto processors. In addition, VPI allows user-level thread systems to schedule new threads in the presence of blocking system calls and page faults. This paper describes VPI and our implementation. We have implemented a complete thread system using VPI, called VPIthreads, and compare its performance to that of current user-level and kernel-level thread systems. Our initial results show the VPI-based thread systems can perform better than current production thread systems.

Keywords: High Performance Computing and Networking, Parallel computing, User-level threads, Linux kernel

1 Introduction

Thread systems are crucial to the performance and correct implementation of language run-time environments, parallel libraries, and parallel applications. However, Linux as well as most operating systems offer only two choices for thread implementation: use the OS supplied implementation of threads, typically Pthreads, or write a thread system from scratch. The former choice is often not suitable for many thread systems due to overhead from unneeded functionality and added code needed to get Pthreads to match the requirements of the application [3]. The latter choice is prohibitive to due the complexity of user-level thread systems and the lack of kernel support.

The problem facing language and library implementors is that there is no middle ground. To address this problem we have developed the virtual processor interface (VPI) for Linux to provide a foundation on which to build custom user-level thread systems with kernel support. Our VPI implementation consists of a small set of kernel modifications and new system calls combined with a small user-level library that provides an interface and execution model that enables the development of arbitrary thread systems. VPI can be used to implement any type of thread system such as threads for a Java virtual machine, OpenMP, or Pthreads (see Figure 1).

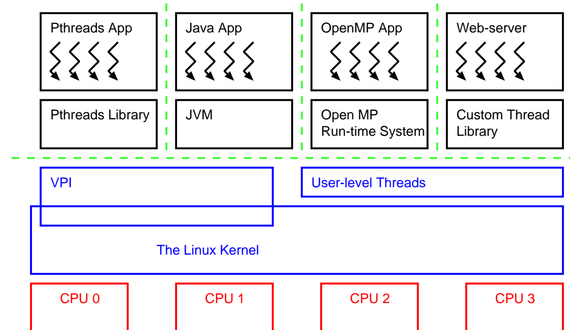

Figure 1. The structure of thread systems built with VPI

To enable kernel coordination with user-level threads, VPI uses an approach similar to scheduler activations [1]. The kernel informs the user-level thread system of scheduling events through an *upcall* mechanism. Unlike scheduler activations, VPI is designed to work in a multiprogramming environment where several multi-threaded applications may be executing simultaneously on all of the processors in a multiprocessor machine. In addition, we have streamlined the upcall mechanism so that kernel contexts are preallocated to handle future upcalls. Our approach to upcalls allow kernel contexts to be used not only to execute user-level threads but also to handle thread system functionality such as timers and signals.

Two key design issues for VPI are portability and flexibility in the programming interface. We achieve portability at the kernel level by modifying only C code portions of the Linux kernel. At the user-level, a context switch mechanism is required. A developer can use either architecture-specific assembly code or portable context switching tech-

niques such as those used in the GNU Pth thread library [6]. Flexibility in the programming interface is achieved using a template approach. The problem is that it is nearly impossible to provide a conventional functional interface to VPI without greatly restricting the way in which thread systems can be implemented because user-level scheduling is tightly coupled with the upcall mechanism. As such, we opt for greater flexibility at the cost of a greater burden on the developer of a user-level thread system. To manage this the complexity of building a new VPI thread system from scratch we have identified common templates that can be as a foundation for new thread systems.

We have constructed a thread system using VPI called VPIthreads and use it to run performance tests. Our experimental results show that the overhead of VPI varies with the amount of resulting upcalls. However, compared to current multiprocessor thread systems VPI performs well on both microbenchmarks and synthetic tests. We show that a simple custom thread system can avoid the overhead in using an OS supplied thread library, but still have effective multiprocessor support.

The rest of this paper is organized as follows. Section 2 provides a brief overview of thread systems and identifies related work. Section 3 describes the virtual processor interface and its execution model. Section 4 provides some details of the Linux implementation of VPI. Section 5 describes the VPIthreads implementation. Section 6 summarizes experimental results that characterize the overhead of VPI and that compare a VPI thread system to other Linux thread systems. Section 7 makes some concluding remarks and gives directions for future work.

2 Background and Related Work

Thread systems can be categorized by their underlying implementation. Broadly, a thread system can be implemented with kernel-level threads, user-level threads or a user-kernel hybrid approach. Kernel-level threads are completely managed by the operating system kernel just like processes except threads that belong to the same process share memory and I/O descriptors. Kernel-level threads have good support for blocking I/O and multiprocessor scheduling because everything is managed by the kernel. However, kernel threads have overhead because switching threads always requires a trap into the kernel.

In contrast, user-level thread systems perform all thread scheduling in user-mode without kernel knowledge. Such thread systems can provide fast thread switching and synchronization, which is useful for applications that require fine-grained threads. Unfortunately, pure user-level threads have no support for multiprocessor scheduling and poor support for blocking I/O operations.

A hybrid approach maps user-level threads onto a set of kernel-level threads. Thus, attempting to achieve the best of user and kernel threads. However, a naive implementation that simply runs user threads on kernel threads can not eliminate the blocking I/O problem and can result in poorer than expected performance because the kernel-level scheduler has no knowledge of the user-level scheduler. A more sophisticated hybrid approach requires coordination between the kernel scheduler and the user scheduler.

The concept of kernel support for user-level threads was pioneered by Anderson et al with *scheduler activations* [1] and Marsh et al with *first-class user-level threads* [12]. These early investigations demonstrated the benefits of user-level thread systems over pure kernel-level threads and the problems that can arise from implementing user-level thread systems on top of kernel-threads without proper kernel support. The fundamental problem is that without kernel support for user-level threads the kernel scheduler may make scheduling decisions without consulting the user-level thread scheduler. Such decisions can result in poor performance in the presence of user-level synchronization or improper execution of user-level threads in the presence of thread priorities.

Some commercial implementations of UNIX adopted the scheduler activations approach, including Digital UNIX [18] and Sun Solaris [13]. Although, the latest version of Solaris, version 9, has dropped support of scheduler activations for a pure kernel-level thread implementation [17]. In addition, scheduler activations have been incorporated into Mach 3.0 [2], BSD/386 [16], SPIN [4, 15], Linux [5], NetBSD [19], FreeBSD [7], and K42 [9]. However, in almost all of these systems the kernel interface to scheduler activations is hidden or only intended to be incorporated into a single user-level thread system. No attempt has been made to expose the scheduler activation interface to developers of thread systems. Therefore, in order to have coordination between user-level and kernel-level threads, the OS supplied thread system must be used.

Good thread support in Linux has developed slowly. The original and still dominate thread system for Linux is LinuxThreads [11], a kernel-thread based implementation of Pthreads. Until recently, two different groups have worked to improve the performance and correctness of Pthreads support for Linux. A group at IBM have developed NGPT (Next Generation POSIX Threading) for Linux [8]. NGPT is a “two-level” thread system based on the GNU Pth thread library [6] that maps user-level threads onto kernel-level threads, although it does not rely on user-kernel coordination. A group at Redhat has developed NPTL (Native POSIX Thread Library) [14], which is an optimized pure kernel-based implementation. Both NGPT and NPTL show significant improvements over LinuxThreads. NPTL is now the standard thread system in Redhat Linux 9. While these projects have improved Linux support for Pthreads, they have not directly improved support for custom user-level thread systems.

3 Interface and Execution Model

The VPI library provides a set of functions used to build thread systems for uniprocessor and multiprocessors. In order to correctly use the VPI functions, it is important to

understand the VPI execution model. The central concept in VPI is the *virtual processor* (VP). VPs are special Linux kernel threads that provide an execution context for user-level threads. In addition, VPs are used by the kernel to issue notifications (upcalls) to the user-level thread system. One of the main responsibilities of a thread system built with VPI is to manage VPs. VPI gives a client thread system complete control over VP execution.

At program startup only one kernel thread is associated with the process. A VPI thread system must invoke the `vpi_init()` function. Among other tasks, `vpi_init()` informs the kernel that this process is a VPI process. This allows the kernel to treat VPs differently than normal kernel threads. The main kernel thread is turned into a VP during `vpi_init()`. The `vpi_init()` function also allows a thread system to create zero or more additional VPs. This is useful if a thread system knows it will need a certain number of execution contexts at startup.

VPs can be in different states. The main VP states are: ACTIVE, REGISTERED, BLOCKED, WAITRESUME, and WAITREGISTER. If a VP is active it means the VP is running or waiting to run on the kernel ready queue. If a VP is REGISTERED, it is blocked in the kernel waiting for an event. The thread system must explicitly register VPs in order to receive notifications from the kernel. A VP enters the BLOCKED state if it issues a blocking system call from the user level or if a blocking page fault is generated. Once the kernel tries to unblock a blocked VP – e.g., if a read request is satisfied – rather than allow it to return to the user-level, it is put into the WAITRESUME state. A VP in the WAITRESUME state must be explicitly resumed by another VP. This functionality gives the thread system complete control over which user-level threads are allowed to execute on the physical processors. Finally, a VP in the WAITREGISTER state is blocked waiting for VPs to become registered. This is need to ensure that initial VPs have registered to receive kernel notifications before starting the application proper.

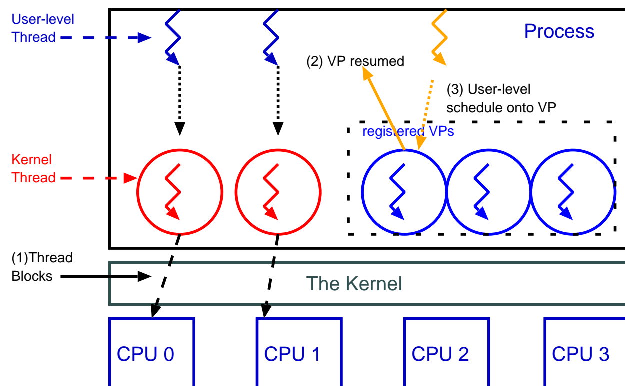

Figure 2. The VP Execution Model

Figure 2 provides a sample snapshot of a thread system in execution. The figure shows that all VPs execute in the same process address space just like kernel threads. Active VPs are used to execute user-level threads. Registered VPs wait to deliver notifications to the user level. When a registered VP wakes up, it returns the type of notification to the user-level. This way, the thread system can determine how to respond to the notification. Figure 2 shows how a user-level thread that blocks in the kernel generates a BLOCKED notification. The notification wakes up a registered VP. This VP can be used by the user-level to schedule a new user-level thread.

Registered VPs can deliver different types of notifications. The supported types are: BLOCKED, UNBLOCKED, PREEMPT, RESUMED, and SIGNAL. As described above, A BLOCKED notification occurs when a VP blocks in the kernel, thus freeing up a physical processor. An UNBLOCKED notification is delivered when a previously blocked VP is unblocked by the kernel. This notification allows the thread system to defer the execution of the unblocked user-level thread until a later time. When a UNBLOCKED notification is delivered, a handle to the blocked VP is passed to the thread system. The thread system can use this handle to explicitly resume the UNBLOCKED thread. The PREEMPT notification is issued each time the kernel preempts an executing VP. The RESUMED notification is delivered when the thread system explicitly resumes a registered VP. The thread system can pass an additional parameter to indicate why the VP was resumed. The SIGNAL notification is used to tell the thread system that a VP was resumed because a signal woke up the VP. The thread system can configure VPI to only deliver certain notifications. For example, it is costly to deliver PREEMPT notifications and not necessary for many applications.

A thread system may need to disable notifications at particular points in a program. For example, inside an idle loop it may be desirable to introduce a small delay with `nanosleep()`. However, if `nanosleep()` blocks, the thread system does not want a notification.s The VPI function, `vpi_kernel_mode()` can be used to change the kernel notification mask at run time.

Using VPs and notifications, a thread system can control the mapping of user-level threads to kernel execution contexts. The thread system can also control the number of active kernel contexts for a particular program instance. Controlling the number of contexts is important to ensure efficient utilization of the physical processors. It is not efficient to have more kernel contexts than actual physical processors. Note that VPI does not give the thread system control over which VPs run on specific processors. Thus, the kernel may still use multiprogramming to run other application in conjunction with one or more VPI applications.

The thread system is responsible for ensuring that there are enough registered VPs that may be needed for notification delivery. On each notification, the thread system is passed the registered VP count. This count can be examined to determine if more VPs should be created and registered.

VPI does not provide an interface to create and manipulate user-level threads. This is left to the thread system to decide how to create and switch between threads. This gives the developer complete control on how to implement threads and, consequently, their granularity. One option is to use existing user-level code such as QuickThreads [10]. VPI provides functionality to associate a user-level thread with a VP. This allows the thread system to find out which user-level thread is running on a particular VP. This is necessary for basic scheduling and synchronization operations. VP thread association can also be used to implement high-performance multiprocessor locks such as those used in the Solaris thread implementation [13].

The complete VPI interface is relatively small. The current exported functions consist of:

```
int vpi_init(int vpcount, void *ptr, vpi_user_func vpman,
             vpi_user_func vpresume, int mask);
int vpi_create(vpi_user_func vpfunc);
int vpi_create_system(vpi_user_func vpfunc);
int vpi_exit(void);
int vpi_finish(int status);
int vpi_get_cpu_count(void);
int vpi_get_vp_count(void);
void* vpi_get_data();
void vpi_set_data(void *);

int vpi_kernel_register(vpi_user_data_t *);
int vpi_kernel_resume(void **);
int vpi_kernel_wait_register(int);
int vpi_kernel_status(void);
int vpi_kernel_mode(int m);

void vpi_lock_init(vpi_lock_t *l);
void vpi_lock_free(vpi_lock_t *l);
void vpi_lock_acquire(vpi_lock_*l);
void vpi_lock_release(vpi_lock *l);
```

4 Implementation

The implementation of VPI consists of kernel modifications and a small user-level library. The user-level library provides wrapper functions to the VPI system calls and manages some for the VP state at the user level. We have incorporated VPI into Linux kernel 2.4.18. However, the changes are minimal, so it is easy to port VPI to newer kernels.

Kernel Modifications One of our goals in modifying the kernel is to keep the changes portable across different processor architectures. As such we only modify the C code. The only files we modify are `include/linux/sched.h`, `init/main.c`, `kernel/fork.c`, `kernel/exit.c`, `kernel/sched.c`. We have added one additional file: `kernel/schedvpi.c` The modifications are designed so that non-VPI applications are not affected by the VPI functionality. We have minimized the amount of code on the critical paths in process management.

The modifications add several new fields to the `task_struct` for VP management. These fields include links for new wait queues used for VP synchronization, state fields, notification type, and a pointer to a per-application data structure called the kernel control block (KCB). A new KCB is allocated for each VPI application. The KCB stores run-time data common to all VPs. The KCB is a shared data structure so it includes a spinlock. Other fields include: a reference count, an registered VP list, a registered VP count, a mode mask, and various statistics variables.

BLOCKED notifications are handled in `do_schedule()`. When a VP (or any kernel thread) blocks, it is put on a wait queue, then `do_schedule()` is called. We check to see if the previous thread is a VPI thread and if it just blocked. If so, a registered VP is resumed to send a BLOCKED notification. PREEMPT notifications are also handled in `do_schedule()`.

UNBLOCK notifications are handled in the `try_to_wake_up()` functions. When a blocked kernel thread is woken up, a call to `try_to_wake_up()` is issued. In this function we determine is the waking thread is a VP. It checks to see if the notification mask is set for UNBLOCK notifications. If so, it wakes up registered VP to delivery the UNBLOCK notification and pass a handle of the unblocked VP to the user-level. The unblocked VP is not really unblocked; it is suspended until the thread system issues a resume system call on the VP handle.

The `kernel/schedvpi.c` file contains all the VP management functions and system calls. It also implements a proc file system entry for debugging and analyzing VPI applications.

User-level Library The user-level part of VPI consists of public functions to be used by a thread system. It also maintains some VP state in the user-level, including a count of currently registered VPs. Each VP is assigned a user-level virtual processor control block (VPCB) for its entire life. The VPCB is used to hold a pointer to the currently executing user-level thread, a pointer to the thread system supplied manager function, and the type of the VP. The user-level code also coordinates program termination and ensures that all VPs are destroyed. Architecture-specific multiprocessor locks are exported to the thread system.

5 VPIthreads

We have implemented a complete thread system using VPI called VPIthreads. We used VPIthreads as a vehicle for developing the VPI interface and to establish some common templates for building new thread systems with VPI. The VPIthreads interface supports many of the core Pthreads functions: thread create, thread join, mutex locks, condition variables, and semaphores. The main features of the current VPI implementation include:

- Multiprocessor support
- Blocking I/O support
- Complete upcall support
- Strict parallelism management

While developing VPIthreads we have identified some core program templates that require a tight coupling of the user-level thread system and VPI. These templates include:

- VP initialization
- VP manager
- VP notification handling
- Thread blocking
- Idle loop

Each of these templates require the thread system to follow VPI conventions, but they also require significant contributions from the thread system based on its requirements. We are working on a documentation scheme that makes it easy to understand how to instantiate the templates.

6 Experimental Results

In order to assess the viability of VPI for building efficient thread systems we have performed some preliminary benchmarking with VPIthreads. We use microbenchmarks to compare basic thread operations between VPIthreads and different Pthreads implementations. We use a synthetic benchmark to reveal the overhead of VPI notifications. All experiments were run on a Dual 1 GHz Pentium III machine with 1 GB of RAM. For VPI, LinuxThreads (LT), and NGPT we used a Redhat 8 installation. For NPTL we used Redhat 9. Note that a direct comparison of VPIthreads to LT, NGPT, and NPTL is not completely fair because VPIthreads is not as sophisticated as these implementations.

Microbenchmarks We consider three microbenchmarks: create/join, condition signal, and barrier. Figure 3 shows the basic performance of executing a create/join pair. The main thread creates a new thread, then immediately joins with the new thread. The new thread simply exits once it executes. LT and NPTL show similar performance because they both must enter the kernel in order to create a new thread. Both VPI and NGPT do a thread create in user space.

Figure 4 shows the cost of synchronizing two threads with condition variables. Again, the user-level implementation, NGPT and VPI perform better than the kernel-level implementations. However, VPI performs worse than NGPT. This is because we have not yet optimized synchronization in our simple thread system. Figure 5 shows the performance of a 32 thread barrier synchronization. The results are similar to the condition variable benchmark.

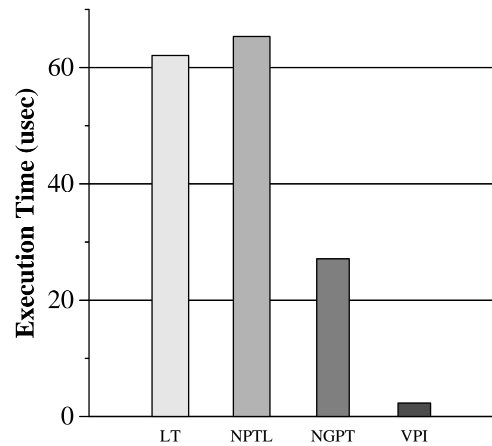

Figure 3. The performance of thread create/join

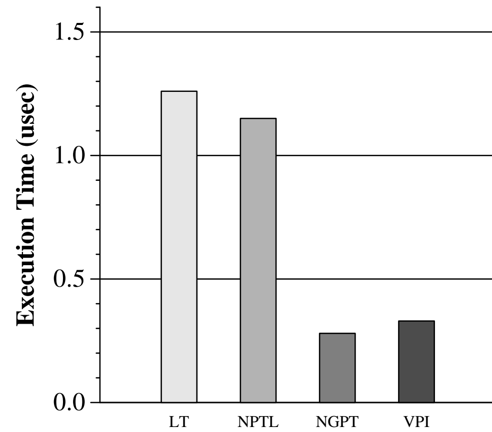

Figure 4. The performance of condition wait/signal

Sun Ping-Pong We used the slightly more sophisticated ping-pong benchmark from Sun Mircosystems [17]. This benchmark is used to determine the scalability and performance of a thread system. The benchmark creates pairs of threads that repeatedly block and unblock each other. The benchmark allows multiple "games" to be executed at the same time. It has been used to benchmark both NGPT and NPTL [8]. Figure 6 shows the results of running the baseline parameters for the ping-pong benchmark. The good performance of VPIthreads is due to its low-overhead user-level context switching. Figure 7 shows the results of running a more complicated instance of the ping-pong benchmark. VPIthreads still performs better than the other thread systems, but not as much as in the baseline test because 8 threads must be multiplexed onto 2 physical processors.

Notification Overhead In order to measure the cost of notifications, we developed a benchmark that simulates an application that computes and issues blocking system calls. A VP performs a fixed amount of computation then issues

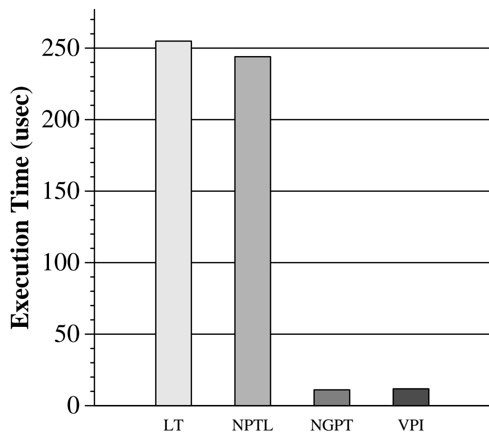

Figure 5. The performance of a 32 thread barrier

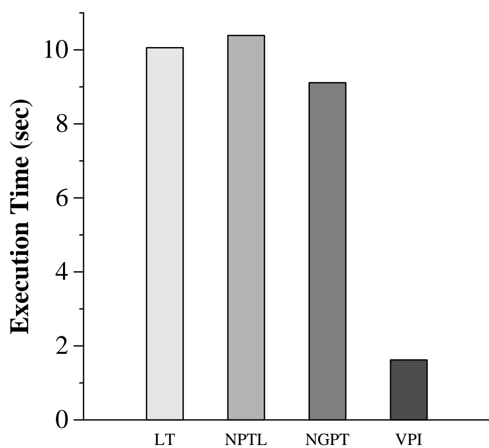

Figure 6. The performance of Sun ping-pong (2 threads, 1 game)

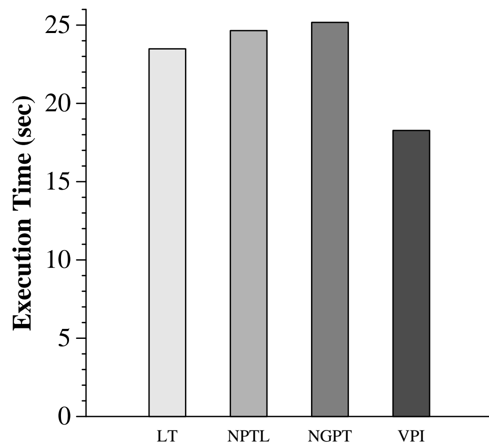

Figure 7. The performance of Sun ping-pong (8 threads, 4 games)

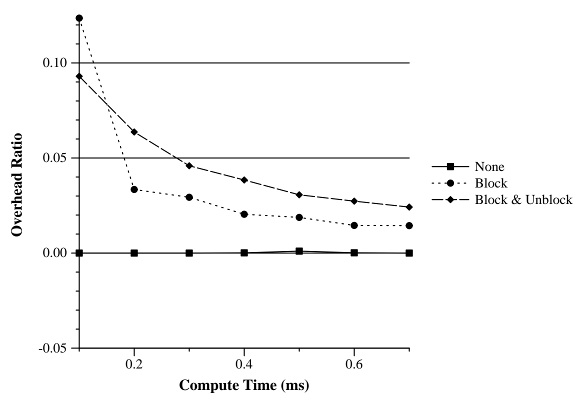

Figure 8. Notification Overhead for 1 pair of threads

a blocking system call. It is then unblocked by a separate kernel thread. Our benchmark can create any number of thread pairs. The overhead benchmark completes a fixed amount of *work* in terms of compute time. We measure the execution time of completing the work with pure kernel threads and no notifications. Then we measure the execution time to complete the same amount of work using VPI with different mode masks: no notifications, BLOCK notifications, and BLOCK and UNBLOCK notifications. We calculate the *overhead ratio* as VPI - KERNEL / KERNEL. In the graphs we vary the frequency of the blocking system calls to show the overhead as a function of notification frequency.

Figure 8 shows the overhead ratios for a single pair of threads. As expected the overhead ratio decreases as the compute time between blocking calls increases as this results in fewer notifications. Real applications will have a considerable amount of computation between blocking calls, thus it is reasonable to expect the overhead from notifications to be small. Figure 9 shows the data for 16 pairs of thread. Notice the absolute values of the ratios are lower because much more work is begin done, and the contribution of the notification time becomes smaller. In addition, more threads means more scheduling. In a real application, other costs will diminish the cost of notifications.

7 Conclusions

In this paper, we described the design, implementation, and performance of the virtual processor interface for building custom, user-level thread systems on Linux. VPI allows developers to build application or language specific thread systems. Our preliminary results show that VPI-based thread systems can perform much better than the stock Linux Pthread implementations. For future work we intend to optimize the VPI notification mechanism, implement high-performance multiprocessor locks, and explore the integration of proportional-share scheduling with VPI and user-level threads.

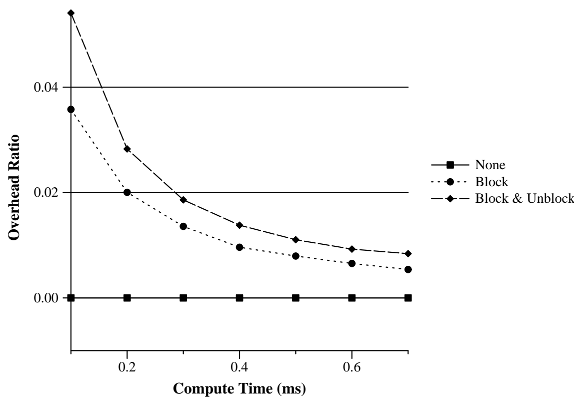

Figure 9. Notification Overhead for 16 pairs of threads

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–70, February 1992.

[2] P. Barton-Davis, D. McNamee, R. Vaswani, and E. D. Lazowska. Adding scheduler activations to Mach 3.0. In *Proceedings of the USENIX Mach III Symposium*, pages 119–136, April 1993. Also as Tech report UW-CSE-92-08-03, University of Washington, Department of Computer Science and Engineering.

[3] G. Benson and R. Olsson. A framework for specializing threads in concurrent run-time systems. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 139–152. Springer-Verlag, Pittsburgh, PA, 1998.

[4] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, December 1995.

[5] Vincent Danjean, Raymond Namyst, and Robert Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proceedings of the 4th Workshop on Runtime Systems for Parallel Programming*, Lecture Notes in Computer Science, Cancun, Mexico, MAY 2000. Springer-Verlag.

[6] Ralf S. Engelschall. Portable multithreading: The signal stack trick for user space thread creation. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[7] FreeBSD. *The FreeBSD KSE Project*, 2003. `http://www.freebsd.org/kse`.

[8] IBM. *Next Generation POSIX Threading*, 2003. `http://www-124.ibm.com/pthreads`.

[9] IBM Research. *The K42 Project*, 2003. `http:/http://www.research.ibm.com/K42`.

[10] D. Keppel. Tools and techniques for building fast portable threads package. Technical Report UW-CSE-93-05-06, University of Washington, Department of Computer Science and Engineering, May 1993.

[11] X. Leroy. *LinuxThreads*, 2003. `http://pauillac.inria.fr/~xleroy/linuxthreads`.

[12] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110–121, Pacifi c Grove, CA, October 1991.

[13] J. Mauro and R. McDougall. *Solaris Internals: Core Kernel Architecture*. Sun Microsystems Press (Prentice-Hall), Palo Alto, CA, 2001.

[14] Redhat. *Native POSIX Threading Library*, 2003. `http://www.redhat.com`.

[15] E. G. Sirer, P. Pardyak, and B. N. Bershad. Strands: An effi - cient and extensible thread management architecture. Technical Report UW-CSE-97-09-01, University of Washington, Department of Computer Science and Engineering, 1997.

[16] Christopher Small and Margo Seltzer. Scheduler activations on bsd: Sharing thread management between kernel and application. Technical Report TR-31-95, Harvard University Division of Engineering and Applied Sciences, 1995.

[17] Sun Microsystems. *Multihtreading in the Solaris Operating Environment*, 2003. `http://wwws.sun.com/software/whitepapers/solaris9/multithread.pdf`.

[18] U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1996.

[19] Nathan Williams. An implementation of scheduler activations on the netbsd operating system. In *Proceedings of the USENIX Freenix Track Technical Program*, Monterey, CA, June 2002.